

PABLO MORENO MARTIN
VÍCTOR ARAGONÉS PIMENTEL
FRANCISCO MORA ORTIZ
EDUARDO LUQUE LÓPEZ
MAGDALENA HUERTAS HANCCO

El ajedrez es uno de los juegos más populares de la historia y se dice que fue uno de los primeros videojuegos. En este curso hemos hecho nosotros del ajedrez nuestro primer videojuego

DISEÑO Y PROGRAMACIÓN DE UN AJEDREZ EN C++



INTRODUCCIÓN

- Un poco de historia para contextualizar:

El ajedrez es un juego de tablero entre dos contrincantes que surge en Europa en el siglo XV a raíz del juego persa shatranj en el que cada uno dispone al inicio de 16 piezas móviles que se colocan sobre un tablero, dividido en 64 casillas o escaques. En su versión de competición, está considerado como un deporte, aunque en la actualidad tiene claramente una dimensión social, educativa y terapéutica.

- En qué consiste:

Se juega sobre un tablero cuadrulado de 8x8 casillas (también llamadas escaques) alternadas en colores blanco y negro, que constituyen las 64 posibles posiciones de las piezas para el desarrollo del juego. Al principio del juego cada jugador tiene dieciséis piezas: un rey, una dama, dos alfiles, dos caballos, dos torres y ocho peones. Se trata de un juego de estrategia en el que el objetivo es «derrocar» al rey del oponente. Esto se hace amenazando la casilla que ocupa el rey con alguna de las piezas propias sin que el otro jugador pueda proteger a su rey interponiendo una pieza entre su rey y la pieza que lo amenaza, mover su rey a un escaque libre o capturar a la pieza que lo está amenazando, lo que trae como resultado el jaque mate y el fin de la partida.

Durante este curso, hemos aprendido a implementar este famoso juego en C++. Hemos tenido que diseñar una estructura lógica para hacer funcionar el juego. Programar los algoritmos de los movimientos, cambio de turno...Hasta llegar a una versión básica pero totalmente funcional y fácil de comprender.

ESTRUCTURACIÓN DEL CÓDIGO.

Diagrama de la máquina de estados:

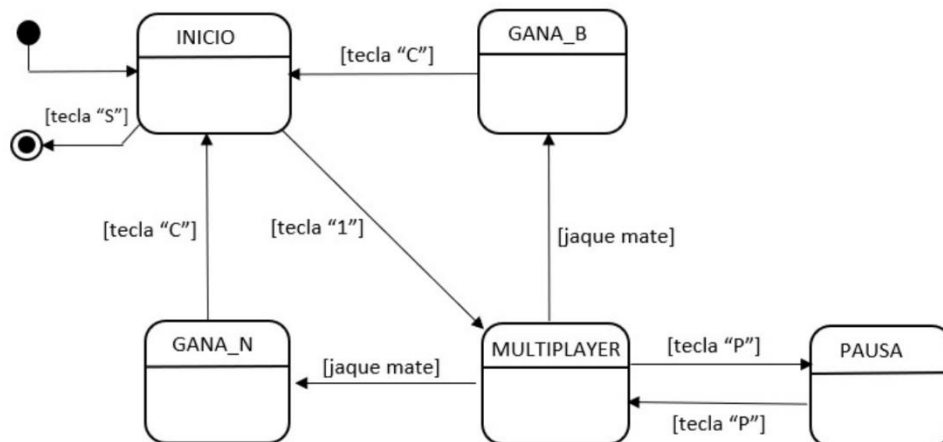


Figura 1.0

Relación entre las distintas clases, las flechas indican las llamadas que se hacen entre ellos:

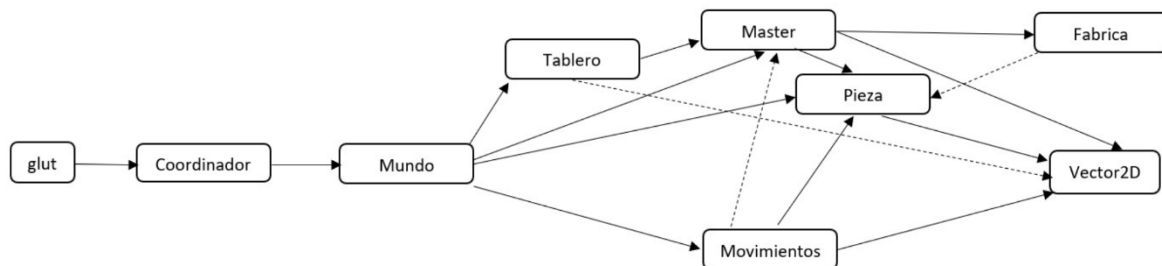


Figura 2.0

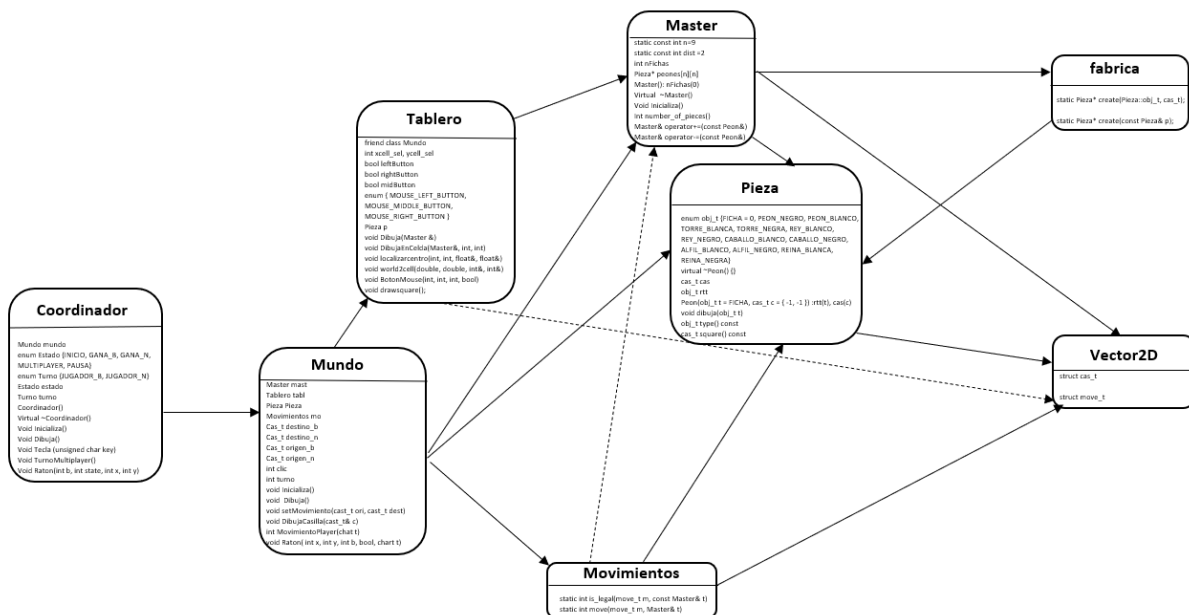


Figura 3.0

MAQUINA DE ESTADOS

En el diagrama de la figura 1.0 se observa la estructuración de la máquina de estados.

CLASES

La relación entre clases es la que se muestra en la figura 2.0, conviene resaltar que la dirección de las flechas indica las llamadas que una clase hace a otra. Así también conviene resaltar que lo que denominamos como glut en el diagrama, son todas aquellas funciones de la librería freeglut.h que empleamos en el fichero de la función principal main (en Ajedrez.cpp) y que a su vez llamarán a la clase Coordinador el cual gestionará el sistema. Más adelante, se procederá a hacer una explicación general de cada uno de los ficheros que contienen las clases así como su relación entre ellos.

Algunos aspectos teóricos a destacar del código de nuestro programa:

HERENCIA:

En el fichero de cabecera Pieza.h, todas las piezas del ajedrez son clases derivadas que heredan de forma public de la clase base Pieza. Esta estructuración se debe a que se identificaron características comunes todas las piezas que se decidieron agrupar en una clase padre o clase case

SOBRECARGA DE OPERADORES Y DE FUNCIONES

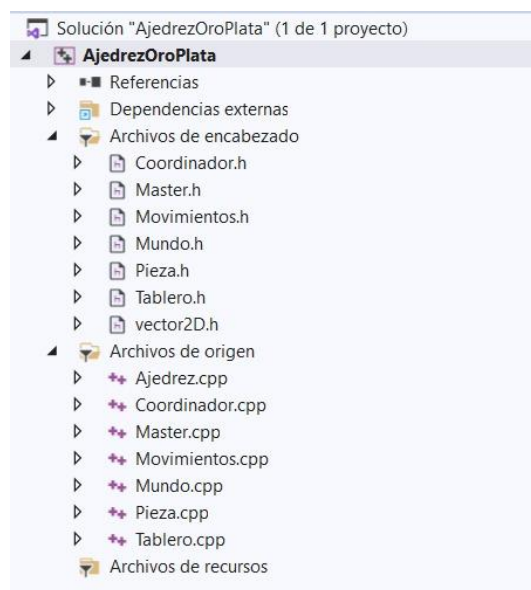
En la clase Master se ha empleado la sobrecarga de dos operadores, en concreto, +=, -=.

```
Master& operator +=(const Pieza&);  
Master& operator -=(const Pieza&);
```

En la clase fabrica de Pieza.h se ha empleado la sobrecarga de funciones empleando argumentos/parámetros distintos.

```
static Pieza* create(Pieza::obj_t, cas_t);  
static Pieza* create(const Pieza& p);
```

IMPLEMENTACIÓN DE LA IDEA



Como se ve en la imagen. Hemos creado varias clases que conforman el diseño del programa. Cada una se encarga de una tarea específica para dar vida al programa final.

VECTOR 2D (archivo .h)

Esta clase tiene dos estructuras, la primera de ellas llamada "cas_t" lo que hace es crear dos variables x e y que almacenarán la información de las coordenadas de las casillas que conforman el tablero, también dispone de una función llamada "print" que se encarga de imprimir por pantalla las coordenadas almacenadas. La segunda estructura llamada "move_t" y contiene la estructura anterior, aprovechándose de sus coordenadas creando un vector2d con el punto de origen y destino.

Conviene resaltar que todas las clases llaman a este fichero, ya que necesitamos hacer constantemente uso de las estructuras que contiene para guardar información sobre las posiciones de nuestras piezas.

TABLERO

Esta clase es la encargada de dibujar el tablero y las piezas con las texturas. La idea es crear el tablero casilla a casilla, se establecen las dimensiones de estas y se dibujan mediante bucles for anidados alternando para obtener una plantilla cuadriculada en blanco y negro. Una vez creado el tablero, el siguiente paso lógico consiste en crear las piezas.

Esta clase llama a la clase master, ya que necesita la información del vector de piezas, peones, así, por ejemplo, dependiendo del tipo de pieza se ejecutará la función DibujaenCelda. O si el puntero de nuestro vector es distinto de NULL significará que en dicha posición se ha creado una instancia de una pieza en esa casilla y por ende debemos llamar a DibujaenCelda, como ocurre en la función Dibuja.

PIEZA

Esta clase la utilizamos para crear todas las piezas del ajedrez. Para nuestro proyecto, hemos utilizado el contenido visto en la teoría denominado como “polimorfismo” y “herencia”. Con la clase pieza creamos las diferentes clases derivadas de dicha clase base. Dentro de la clase Pieza.h establecemos una enumeración donde indicaremos el tipo de pieza y al que nos referiremos a lo largo del código ayudándonos de los switch case. Conviene también mencionar que esta clase padre contiene una función dibuja que llama a todos los dibuja() de las clases hijas.

Valiéndonos de la derivación, planteamos así la estructuración/creación de las piezas, porque creemos que de este modo resulta más sencillo comprender y analizar la dinámica del programa. Además, todas las piezas del juego de ajedrez tienen en común los atributos y métodos que hemos identificado como comunes y que posteriormente hemos implementado en la definición de la clase Pieza, de allí la necesidad de aplicar el concepto de herencia.

MOVIMIENTO

Se definen los movimientos correspondientes a cada pieza, la legalidad de estos, los movimientos de comer y los límites del tablero para cada pieza. Esta clase contendrá dos funciones con la que estableceremos los límites de movimiento de cada pieza.

```
class Movimientos {
public:
    static int is_legal(move_t m, const Master& t);
    static int move(move_t m, Master& t);
};
```

Con la primera, verificaremos que todos los movimientos sean legales o permitidos. Por ejemplo, reflejaremos la imposibilidad de matar piezas de nuestro propio equipo. También estableceremos lo que ocurre en un posible jaque mate, o cualquier posible situación que pueda darse durante el juego. El código está estructurado y elaborado a través de funciones que retornan valores int. Con esto crearemos la red de posibles sucesos con el que nos moveremos de un .cpp a otro según el juego lo necesite. Por ejemplo, la función MovimientoPlayer de la clase Mundo, usa dichos valores de retorno de tipo int para clasificar:

1. si hago un movimiento a casilla vacía, 2. si como, o 3. si quiero hacer un movimiento no válido y por ende continuo con mi turno, estas clasificaciones devuelven a su vez un return de un valor

entero a la función TurnoMultiplayer del Coordinador, que se encarga de hacer la ejecución de cada una de las tres opciones antes mencionadas, así como de evaluar si alguno de los jugadores gana.

COORDINADOR

Es el encargado de la coordinación de la máquina de estados que maneja el juego. Con él gestionamos el menú de inicio, las funciones extra como la pausa, los estados de victoria o los cambios de turno, retorno al inicio tras una victoria. Esta herramienta ha sido vista en la última sesión del laboratorio de la asignatura, aunque el concepto de máquina de estados nos es familiar gracias a nuestra participación en otras asignaturas de la carrera. Los sonidos de impacto y la música están incorporados en la máquina de estados.

AJEDREZ (solo .CPP)

Es el código principal del juego, en él se encuentra la función principal, allí también es donde se establecen las funciones proporcionadas por la librería FREEGLUT.H. Con este archivo, el tiempo en el juego correrá favoreciendo el movimiento. También es el encargado de manejar el ratón, el teclado o de dibujar todo lo programado en otros archivos.

MUNDO

Como hemos visto en los laboratorios de la asignatura. Es la encargada de proporcionar al proyecto una funcionalidad que finalmente vemos por pantalla haciendo referencia al resto de clases. El “Mundo” también es fruto de las sesiones del laboratorio donde nos iniciamos a la orientación a objetos en el lenguaje C++. Este archivo es imprescindible para la implementación del código.

A través de incluir los ficheros de cabecera del resto de clases, creamos instancias de las clases llamadas, como atributos privados; así, llamamos a nuestro Tablero para que se dibuje, a nuestro vector de piezas contenido en Master, a los Movimientos para que se evalúe su legalidad y el tipo de movimiento que el usuario quiere hacer, a las estructuras de cas_t para que almacenemos en ellas el origen y el destino que el usuario seleccione con el mouse, de las piezas de ambos bandos.

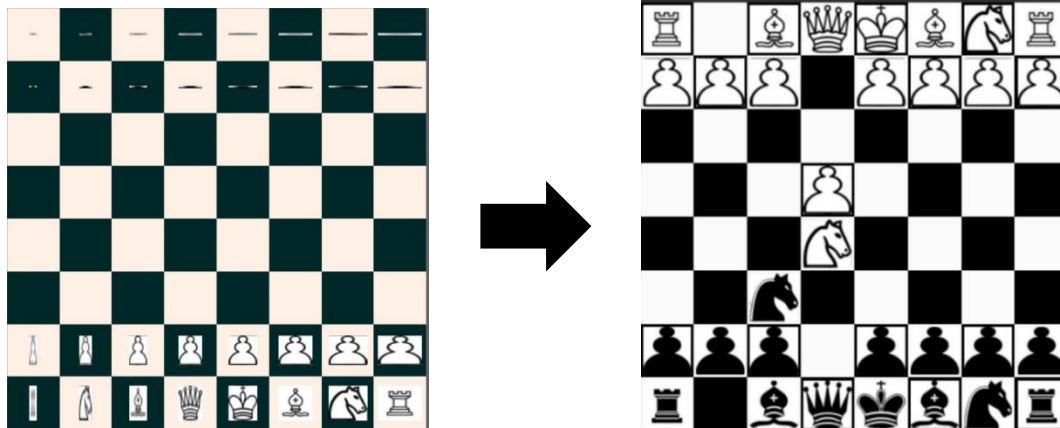
MASTER

Será el encargado de poner las casillas a cero (utilizando NULL) para evitar el depósito de contenido basura en su interior. También es el encargado de inicializar las piezas en el tablero determinando que casilla ocupa cada pieza de nuestro ajedrez. Contaremos las piezas, y sobrecargaremos dos operadores (+= y -=) para la creación y destrucción de las distintas piezas en los movimientos simples y de comer.

Esta clase llama a su vez a la clase Pieza para crear un vector de piezas, inicializando los punteros a NULL en su constructor, creando a través de memoria dinámica las instancias a las piezas, o eliminando a través de un delete las piezas a medida que desaparezcan ya sea por ser comidas o porque se mueven de casillas y necesitan ser redibujadas y ubicadas en una nueva posición en nuestro vector de piezas.

EVOLUCIÓN DE LA TEMÁTICA

Comenzamos con las primeras iteraciones tratando de representar el tablero con unas imágenes de piezas básicas para seguir trabajando en el modelo, sin embargo, al tratar de representarlas ocupando un espacio cuadrado del tablero, las imágenes se mostraban más reducidas conforme se ascendía en el tablero:



Posteriormente conseguimos realizar adecuadamente la representación de las piezas para que ocupen su proporción de la casilla.

A partir de este punto decidimos conseguir figuras personalizadas de la temática de físicos vs ingenieros, donde conseguimos la siguiente distribución, y finalmente, para añadir mayor limpieza a la partida, procedemos a eliminar el fondo de estas imágenes añadiendo un canal alfa con las funciones incluidas en la librería freeglut:



En el bando de los físicos tenemos los siguientes personajes: Sheldon Cooper(de The big bang Theory) como torre, uamito la mascota de la UAM como caballo, Michael Faraday como alfil, Marie Curie como reina, Albert Einstein como rey y los átomos como peones.

En el bando de los ingenieros tenemos a Howard(de The big bang Theory) como torre, la mascota de la ETSIDI(un oso) como caballo, a Elon Musk como alfil, a Hedy Lamarr como reina, a Nikola Tesla como rey y a unos engranajes como peones.

DESAFIOS Y SOLUCIONES PRESENTADOS

- Uno de los problemas principales de hallar el código era el movimiento de las piezas. Sin duda fue el punto de inflexión del proyecto donde más tiempo invertimos. Salía constantemente una interrupción cada vez que movíamos una ficha en la creación y destrucción de fichas en la parte final del Movimientos.cpp. Tuvimos que borrar parte del código e ir desde lo más simple avanzando poco a poco. Finalmente solucionamos el error que pudo costarnos la entrega del trabajo a tiempo.
- Otro de los problemas que surgió durante la entrega fue la representación de las imágenes, tratando de representarlas con un fondo sin color añadimos una versión de las imágenes con un fondo transparente añadido a través de un programa externo para tratar de representarlas sin un fondo blanco, sin embargo, el programa seguía compilando de igual forma ya que añadía este fondo de forma automática. Tratamos de resolver el problema cambiando el formato de la imagen, aunque en realidad, el png es el adecuado para la representación sin fondo, por ello pensamos que probablemente debíamos encontrar una función existente a ser posible en la librería de freeglut para poder añadir imágenes con fondo transparente, y después de varias pruebas con funciones posibles encontramos una con la que efectivamente ya no añadía un fondo blanco a la imagen.
- Otro problema que tuvimos que afrontar fue que al principio la torre blanca de arriba a la izquierda estaba situada en la casilla [0,0] funcionaba mal ya que referenciamos todo a [0,0] y lo solucionamos situando la casilla [0,0] en la casilla [1,1] y así se deja la casilla [0,0] libre evitando errores.
- Otro problema sucedido está relacionado con el caballo blanco situado en la casilla [1,2] ya que no se movía y además cambiaba el turno de la partida ya que no tenía límites de movimiento, es decir, que se podría salir del tablero. Además, tampoco funcionaba la función en mundo llamada "DibujaCasilla", es decir, que la casilla no se seleccionaba cuando tocábamos ese caballo, lo que hicimos para solucionarlo fue ponerles límites a los movimientos del caballo.
- Otro problema tenía relación con el audio, ya que tratando de disponer de música orientada a la duración de la partida y el menú queríamos que se repitiera la canción en bucle hasta que saliéramos del estado correspondiente, para ello tuvimos que buscar una función

distinta a la empleada en el laboratorio, ya que esta repite el sonido de forma completa sin posibilidad de corte y una sola vez, por ello tras iterar con diversas funciones de la librería, encontramos las funciones “playmusica” y stop música, en las que, habilitando un true como segundo argumento, podremos conseguir que este audio se reproduzca de forma indefinida hasta activar la función “stopmusica”.

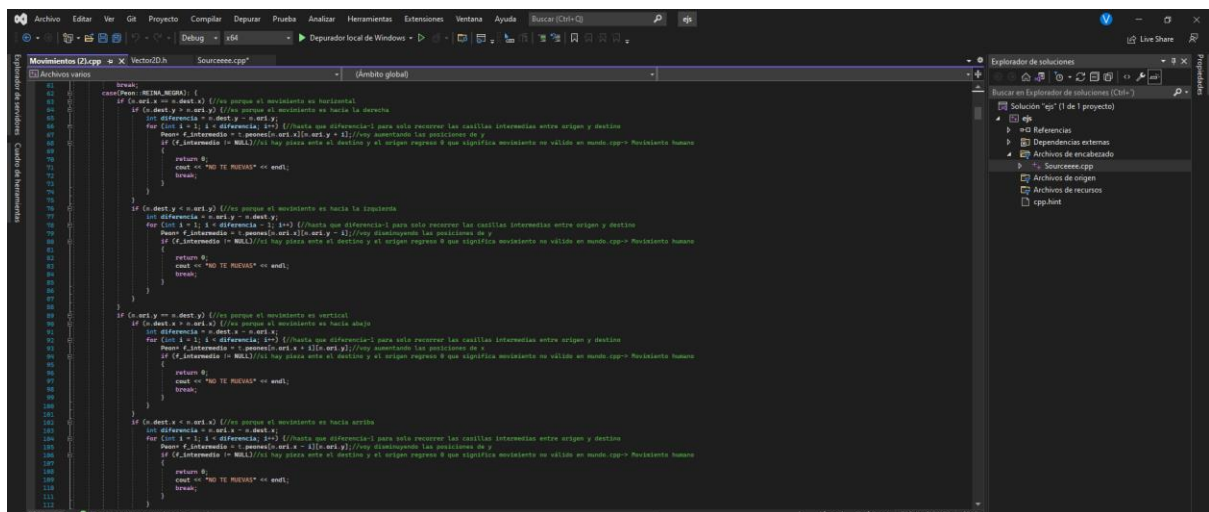
POSIBLES MEJORAS Y SOLUCIONES

A pesar de que hemos conseguido un buen nivel de usabilidad, si que existen posibles mejoras que se podrían implementar en un futuro.

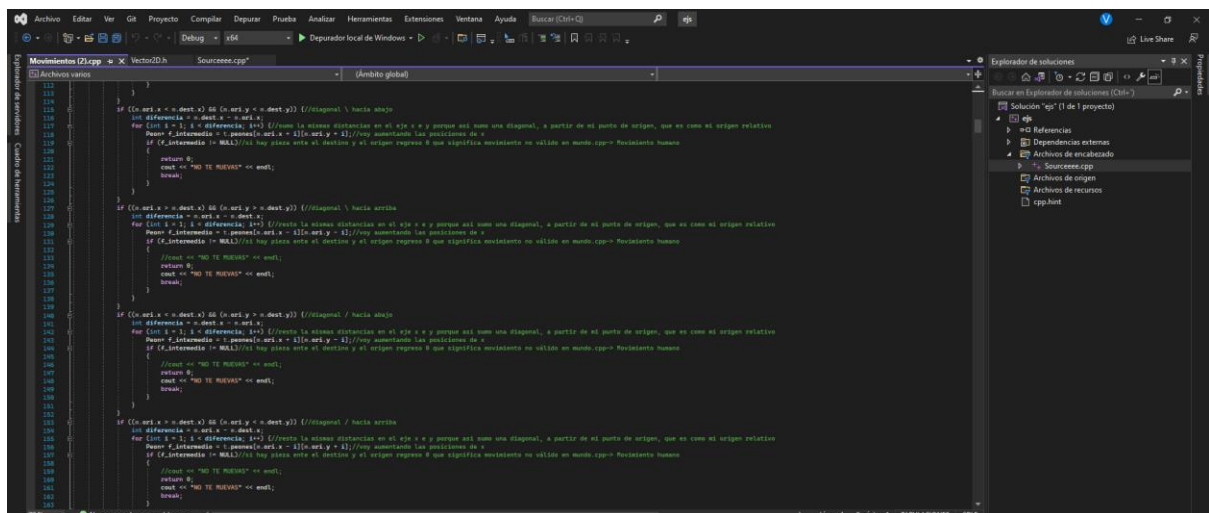
La más importante es tratar de evitar que las piezas se salten unas a otras. Como sabemos, en el ajedrez está prohibido llevar una de tus piezas a una casilla si en el camino se encuentra a otra pieza. Tanto si es una pieza del mismo color como si es una del contrario, en cuyo caso solo podremos comernos a la pieza enemiga.

La lógica para este problema sería analizar las casillas existentes entre la de origen y la de destino y estudiar si alguna de estas está ocupada para validar o no el movimiento. Pero no hemos sabido acoplarlo al proyecto de manera correcta, aunque si llegamos a realizar un análisis lógico e implementación de una posible solución de código. Pero este no se muestra en el proyecto final que presentaremos en nuestra entrega.

Este es uno de los principales prototipos de código implementado para su solución:



```
break;
case (Dest == DestA, DestB): {
    if (x.dest.x == x.dest.y) { //no porque el movimiento es horizontal
        if (x.dest.y == x.dest.x) { //no porque el movimiento es hacia la derecha
            int diferencia = x.dest.y - x.dest.x;
            for (int i = 1; i < diferencia; i++) { //hasta que diferencia para solo recorrer las casillas intermedias entre origen y destino
                int casilla_intermedia = x.dest.x + i; //no aumentamos las posiciones de y
                if (x.intermedia != NULL) { //si hay pieza entre el destino y el origen regresa 0 que significa movimiento no válido en mundo.cpp= Movimiento Numero
                    return 0;
                    cout << "NO TE PUEDES" << endl;
                    break;
                }
            }
        }
        if (x.dest.y < x.dest.x) { //no porque el movimiento es hacia la izquierda
            int diferencia = x.dest.y - x.dest.x;
            for (int i = 1; i < diferencia; i++) { //hasta que diferencia para solo recorrer las casillas intermedias entre origen y destino
                int casilla_intermedia = x.dest.x + i; //no aumentamos las posiciones de y
                if (x.intermedia != NULL) { //si hay pieza entre el destino y el origen regresa 0 que significa movimiento no válido en mundo.cpp= Movimiento Numero
                    return 0;
                    cout << "NO TE PUEDES" << endl;
                    break;
                }
            }
        }
    }
    if (x.dest.y == x.dest.x) { //no porque el movimiento es vertical
        if (x.dest.x == x.dest.x) { //no porque el movimiento es hacia abajo
            int diferencia = x.dest.x - x.dest.x;
            for (int i = 1; i < diferencia; i++) { //hasta que diferencia para solo recorrer las casillas intermedias entre origen y destino
                int casilla_intermedia = x.dest.x + i; //no aumentamos las posiciones de y
                if (x.intermedia != NULL) { //si hay pieza entre el destino y el origen regresa 0 que significa movimiento no válido en mundo.cpp= Movimiento Numero
                    return 0;
                    cout << "NO TE PUEDES" << endl;
                    break;
                }
            }
        }
        if (x.dest.x > x.dest.x) { //no porque el movimiento es hacia arriba
            int diferencia = x.dest.x - x.dest.x;
            for (int i = 1; i < diferencia; i++) { //hasta que diferencia para solo recorrer las casillas intermedias entre origen y destino
                int casilla_intermedia = x.dest.x + i; //no aumentamos las posiciones de y
                if (x.intermedia != NULL) { //si hay pieza entre el destino y el origen regresa 0 que significa movimiento no válido en mundo.cpp= Movimiento Numero
                    return 0;
                    cout << "NO TE PUEDES" << endl;
                    break;
                }
            }
        }
    }
}
```



Este código compilaba, y funcionaba. El principal problema es que creaba otros problemas en la movilidad de la reina que perjudicaban más que ayudar. Por esta razón decidimos eliminarlo del código entregado.

La principal mejora que nos hubiese gustado entregar, es la implementación de una inteligencia artificial con un modo “SINGLEPLAYER”. Realmente, era nuestro objetivo desde el primer momento, aunque los consecuentes problemas en el código, junto a un cuatrimestre difícil, nos ha impedido conseguir una correcta implementación de la IA.

Otras posibles mejoras podrían ser implementar diversas jugadas conocidas, tales como la apertura siciliana, captura al paso o el jaque mate (Ya que en el ajedrez esto se considera realmente como victoria, El rey no puede ser comido) En cuanto a las otras jugadas, son jugadas que solo se pueden realizar en momentos muy concretos de la partida. Por lo cual, para implementarlas, se debe estudiar la situación de las piezas de manera individual para corroborar si esas jugadas son legales o no. Para dar valor a esto, se podrían hacer variaciones por pantalla, o algún tipo de “bonus” que se vea en pantalla cada vez que se realice una de estas jugadas.

CONCLUSIÓN

Se concluye que los resultados que hemos obtenido, en cuanto al producto final del juego, son satisfactorios, puede que no se hayan conseguido un juego impoluto al 100%, pero hemos hecho todo lo que estuvo a nuestro alcance dentro de nuestras posibilidades de tiempo y conocimientos adquiridos a lo largo de la asignatura

Conviene resaltar que el desarrollo del programa supuso todo un reto, ya que nos topamos con contantes problemas que no solo requerían tener claros los conceptos teóricos, sino que también exigían un toque de creatividad por parte de los miembros de nuestro grupo.

Con todo lo ya comentado, damos por finalizado el trabajo propuesto de esta asignatura.